

Logic Programming Engineering – Assignment 4

Dr.-Ing. Sascha Klüppelholz

Exercise 4.1 [Reimplementation of list operations]

Reimplement the following list operations and add all predicates to `mylist.prolog`.

- a) Provide a Prolog predicate for computing/ deciding the length of a given list.

Solution:

```
mylist_length([],0).
```

```
mylist_length([_|T],N) :-  
    mylist_length(T,N1),  
    N is N1+1.
```

- b) Write a predicate `mylist_member/2` that decides membership of an element to list.

Solution:

```
mylist_member(Element, [Element | _]) :- !.
```

```
mylist_member(Element, [_ | Tail]) :-  
    mylist_member(Element, Tail).
```

- c) Write a predicate `mylist_occurrences/3` that counts the number of occurrences of an element within a list.

Solution:

```
mylist_occurrences(_, [], 0) :- !.
```

```
mylist_occurrences(Element, [Head | Tail], Occurrences) :-  
    mylist_occurrences(Element, Tail, O2),  
    (  
        (Head = Element, Occurrences is O2+1, !);  
        (Head \= Element, Occurrences is O2)  
    ).
```

- d) Write a predicate `mylist_last/2` that returns the last element of a list.

Solution:

```
mylist_last(LastElement, [LastElement]) :- !.
```

```
mylist_last( Element, [_ | List]):-  
    mylist_last( Element, List).
```

Alternative solution based on length/2 and nth1/3:

```
mylist_last (LastElement , List) :-
    length (List , Len),
    nth1 (Len , List , LastElement).
```

- e) Provide your own predicate mylist_element_at/3 which returns the n-th element of a list. The index of the head of a list should be 1.

Solution:

```
mylist_element_at ([Head | _], 1, Element) :-
    Element = Head , !.

mylist_element_at ([_ | Tail], I, Element) :-
    IPrime is I-1,
    mylist_element_at (Tail , IPrime , Element).
```

- f) Write a Prolog predicate mylist_add/3 that allows adding an element to a list without introducing duplicates. Check what Prolog returns for the following query: mylist_add(a,[a,b,c,a],[a,a,b,c,a]).

Solution:

```
mylist_add (Element , List , List) :-
    member (Element , List) , !.

mylist_add (Element , List , [Element | List]).
```

- g) Provide a Prolog predicate mylist_remove/3 that allows to remove all occurrences of an element from a given list.

Solution:

```
mylist_remove (_, [], []) :- !.

mylist_remove (X, [X | Tail], Result) :-
    !, mylist_remove (X, Tail , Result).

mylist_remove (X, [Y | Tail], [Y | Result]) :-
    X \= Y, !,
    mylist_remove (X, Tail , Result).
```

- h) Provide a predicate mylist_remove_duplicates/2 for removing duplicates from list.

Solution:

```
mylist_remove_duplicates ([], []).

mylist_remove_duplicates ([Head | Tail], Result) :-
    member (Head , Tail) , !,
    mylist_remove_duplicates (Tail , Result).

mylist_remove_duplicates ([Head | Tail], [Head | Result]) :-
    mylist_remove_duplicates (Tail , Result).
```

- i) Write a predicate `mylist_replace/4` that allows replacing all occurrences of an element in list with another element.

Solution:

```
mylist_replace([],_,_,[]).
```

```
mylist_replace([Head | Tail1], Element, NewElement, [NewElement | Tail2]) :-  
    Head = Element, !,  
    mylist_replace(Tail1, Head, NewElement, Tail2).
```

```
mylist_replace([Head | Tail1], Element, NewElement, [Head | Tail2]) :-  
    mylist_replace(Tail1, Element, NewElement, Tail2).
```

- j) Provide a predicate `mylist_concat/3` for list concatenation.

Solution:

```
mylist_concat([], List, List).
```

```
mylist_concat([Element | List1], List2, [Element | List3]) :-  
    mylist_concat(List1, List2, List3).
```

- k) Write a Prolog predicate `mylist_reverse/2` for reversing a list. Define a predicate `mylist_palindrome/1` on the basis of `mylist_reverse/2` that decides whether a given list is a palindrome.

Solution:

```
mylist_reverse([],[]).
```

```
mylist_reverse([Head | Tail], RList) :-  
    mylist_reverse(Tail, RevTail),  
    append(RevTail, [Head], RList).
```

Alternative solution using an accumulator:

```
mylist_reverse2(List, RList) :-  
    mylist_reverse2(List, [], RList).
```

```
mylist_reverse2([], Accumulator, Accumulator).
```

```
mylist_reverse2([Head | Tail], Accumulator, RList) :-  
    mylist_reverse2(Tail, [Head | Accumulator], RList).
```

```
mylist_palindrome(L):- mylist_reverse(L,L).
```

Look at the internal implementation within SWI-Prolog of the above predicates by using `listing/1`.

Exercise 4.2 [Powerset]

Provide a predicate `mylist_power/2` for creating the powerset of a list (interpreted as set).

Solution:

```
mylist_power([], []).
```

```
mylist_power([Element | Rest], PowerSet) :-  
    mylist_power(Rest, PowerWOE),  
    findall([Element | Tail], member(Tail, PowerWOE), PowerWE),  
    append(PowerWOE, PowerWE, PowerSet).
```

Alternative solution using own implementation of subset (without cut)

```
mylist_subset([], []).
```

```
mylist_subset([_ | Tail], P) :-  
    mylist_subset(Tail, P).
```

```
mylist_subset([Head | Tail], [Head | P]) :-  
    mylist_subset(Tail, P).
```

```
% retract old subset predicate and replace with own implementation  
:- retractall(subset(_,_)).  
subset(X,Y) :- mylist_subset(X,Y).
```

```
mylist_power(S,P) :-  
    findall(X, subset(S,X), P).
```

Exercise 4.3 [List permutations]

Provide a predicate `mylist_permutation/2` that enumerates all permutations of a list.

Solution:

```
mylist_permutation([], []).
```

```
mylist_permutation(List, [Element | Permutation]) :-  
    select(Element, List, Rest),  
    mylist_permutation(Rest, Permutation).
```